An Analysis of Software Testing

Steven Hepting
Electronics Systems Engineering
200 238 691

Work Term 2

May 12, 2006

"Good software testing can increase your productivity, improve your designs, raise your quality, and make you more productive overall." - Chromatic (O'Reilly)

Contents

1 Executive Summary

This document is an analysis of the utility of software testing in the scope of an organization. It details the importance of testing, the options for creating test programs, and finishes of suggesting several options for an organization looking to start a formal testing program.

The first thing to understand is *"Why testing is important"*. In a psychological sense, testing creates scope boundaries, and creates clear goals for programmers. In a logical sense, it gives managers a lot of flexibility because of the progress information it provides. They can see which tests are passing and then use that information to make informed decisions regarding schedule and labour allocation. In an economic sense, testing mitigates the costs associated with many pitfalls common to software development. Although it does require investment in both prevention and status appraisal, the reduction of internal and external costs of failure almost always outweigh the initial outlay in financial terms.

The next step is logically to ask *"What can I do about it?"* The most basic answer would be to write tests for each part of the program that has some functionality. These tests are called "functional tests" and tests specific aspects of a program. "Domain tests" could also be used to check each aspect, but the input now consists of a range of values in order to reveal more insidious errors. When the program under discussion is being written for a third party, "specification-based" testing becomes very important to ensure every aspect of the program specification works as specified. While a program is under development, "regression testing" is very valuable to run regular checks ensuring the functionality of the program does not break when new changes are made. The last type of testing discussed in this document is "high-volume automated testing". This testing does not just test functionality but runs many tests at once, in succession (or in parallel) to check for infrequent errors or spurious results that only appear under high load situations.

This paper will now clearly elucidate the reasons for testing, and will then provide a series of steps which will exemplify a process to follow when creating your own testing framework.

2 Introduction

2.1 Testing & the Program Specification

Often, when referring to software testing in a logistical sense, something called a *functional specification* is commonly talked about. This is a document that describes in detail, each function of the proposed software project. It is often part of the agreed upon contract between parties before work is started on a programming project.

The IEEE/IFIP definition of failure is "A system not performing up to its specification." As such, it's clear that with no specification, it's difficult to know when a test has failed. Not only does a functional specification bring clarity and direction to a group, but it also allows progress to be charted.

Ram Chillarege explains, "the advantage of having a functional specification is that the test generation activity could happen in parallel with the development of the code."[1] Not only does this allow the serialization bottleneck to be removed from the code development, but it also forces a clarity of goals (as was already talked about), and provides documentation for the customer. Even if there is no customer (it's just an in-house software project) the first two advantages outweigh any other objections.

The functional specification and testing are an important duo with respect to software development. Their relationship can be related to that of a Mob-boss and his henchmen. The Mob-boss says how things are going to be, and the henchmen enforce that view. The make sure everything fits that view. In the software world, the functional specification says how the program is going to be, and the testing of the project points out every place where the program isn't following that specification. Although the functional specification is very important, this document will only be focusing on the software testing part of the equation.

2.2 Formatting and Design

Any code in this document with be displayed in a `monospaced font` to make the distinction between computer code and regular prose.

The document has been typeset with LaTeX, the *de facto* standard of typesetting in academia. It changes `cite{kaner03}` to [2] and creates very nice equations:

$$\left|\sum_{i=1}^{n} a_i b_i\right| \le \left(\sum_{i=1}^{n} a_i^2\right)^{1/2} \left(\sum_{i=1}^{n} b_i^2\right)^{1/2}$$

2.3 Audience

Throughout this report, testing will be dealt with in terms of its utility in the software world. However, its use extends far beyond software systems and their development. It is immediately useful for product designers to have a prototype to assert their hypotheses and for chemical engineers to get a pre-release sample of their new formula. Despite, the wide range of the topic, I will just drill down into the software world and it is left to the reader to make transference connections to their own field.

2.4 About PMC-Sierra

PMC-Sierra is not a software company *per se* but they do have a software division to create tools for the rest of the department. The actual products that PMC make are networking chips for enterprise switching hardware. That means they design the microchips to go into large network switching equipment for Internet Service Providers and Telcos.

In terms of software, the situation is made interesting because of the number of locations PMC-Sierra has. They have offices in Saskatoon, Winnipeg, Montréal, Ottawa, Burnaby, Santa-Clara, Jamestown, Ireland, Shanghai, Beijing, Seoul, Bangalore, and most recently Israel. All software being developed is kept in online repositories to allow employees at all locations to obtain the newest versions. These repositories (sometimes

called *revision control systems*) also allow developers to work collaboratively on software programming. They can *check out* a version of a file, make changes to it, test it, and check it back in once they know it works.

PMC-Sierra creates software merely to add value to the microchips they sell. Their customers are not the general public but, companies who build enterprise networking equipment.

3 Why is it important?

> *"Increasingly, people seem to misinterpret complexity as sophistication, which is baffling — the incomprehensible should cause suspicion rather than admiration. Possibly this trend results from a mistaken belief that using a somewhat mysterious device confers an aura of power on the user."*
> - Niklaus Wirth

"Why is testing important?" This is the first question that any programmer, or person who manages programmers, should asking. After all, if testing *is not* important, there isn't any sense in studying the intricacies of the different types of testing or the best places to use them. If testing is not important, it would actually be a poor choice of one's time to spend it reading about testing.

However, software testing *is* important. Software is complex and testing provides many benefits. The rest of this section will elucidate the psychological, logistical, and economic reasons why.

3.1 Psychologically

3.1.1 Scope Boundaries

The presence of both a *functional specification* and a *test suite* for a program keep the scope of a program in check. *Scope Creep* is a problem many companies have that occurs when a software project grows beyond its originally intended size. It often happens a small bit at a time, and isn't until the group looks back that they realize how much the program has changed.

The problem can be mitigated by an accurate functional specification along with tests written to determine the current compliance of the program to that specification. This approach works because it allows developers and managers to refer to the functional spec each time a change is though about. The testing part of the process allows the software to be regularly checked to ensure it is progressing in sync with the current development plan. The tests are written to compare the *actual* functionality to the *desired* functionality. The results can be used to determine what needs to be improved or changed to achieve that compliance.

Scope boundaries also apply to groups within an organization to determine what they are each responsible for. Each group need not worry about the others since they can read in the functional spec how the other parts of the program will interact with theirs. This not only prevents separate groups from doing the same work, but it also removes the programming serialization bottleneck.

During my work-term I saw an example of scope boundaries failing. Two groups of employees from different locations were getting together at teleconference meeting to discuss creating a testing framework to check new releases of software to ensure they would work on all hardware versions. As the discussion continued, it became clear that the two groups had each been working on their own program to do the job a different way. The irony was that each group had knowledge of the other's efforts in this direction. If there had been a functional specification written each group could have each worked on their part of the project, and work could have progressed much more quickly.

3.1.2 Clear Goals

Software tests allow a programmer to reduce redundancy in the development process. A first-time programmer will write some code, then check to see if it does what they want. Then they change the code and check it again. If they decide to come back later and change a few lines, they have to figure out what the code does and then figure out how to test it again. A more mature programmer will take that action of checking the code, and put it into a test. Then, *every time* they want to make a change, they can use that same test to ensure the piece of code still functions correctly. This reduces redundancy for the programmer.

The creation of tests also allows more detailed testing to be done. A regular test, will test the functionality of some code. A domain test however, will take that test and run it several times with different values, allowing a more thorough testing to be performed without having to run the program manually many times after each change. This process doesn't even require any more effort than manual testing since the test only needs to be written once, and can then be run many times.

Regular testing allows a programmer to move on to implementing the next feature. This is a mature form of development, as opposed to writing features and trying to envisage how they will fit together. David Allen explains in his book "Getting Things Done" that people can keep only a few things on their mind at once. His suggestions boil down to storing information in a readily available place to get it off your mind. This applies to the world of software development but writing tests. A saved test allows the implementation of the code to be forgotten about, without any ill effects. This allows programmers to move on to getting new work done.

Preventing redundant work is not only important to writing good software. It's also important for *retaining employees*. At PMC-Sierra I had two brilliant co-workers (one was actually my supervisor) who got amazing amounts of work done, and who were generally model employees. As it turns out, they had both become fed up with the egregious testing practices of their previous employer and had left come to work at PMC. A loss of two excellent employees would be a blow to any employer but in a company like theirs where no records were kept, the loss of two knowledgeable employees would have been devastating.

3.2 Logistical

Testing allows companies to make informed logistical decisions. It allows for a status check on the progress of a project which, in turn, allows for good planning decisions to be made. The value of this section is weighted heavily towards management, with very little value for individual developers.

3.2.1 Schedule

Software testing allows managers to make informed decisions regarding the schedule of a project. Often, managers are just left with vague progress reports by programmers, or (even worse) reports of how many lines of code have been written. These sources are very poor indicators of progress being, at best, wildly inaccurate.

In many organizations release delays are almost a given. That means the product is not ready by the time that management projected they would be done. The problem stems from the uniqueness of software development as a whole. One part of the program slips and everything else is held up. Or, a big problem is found just a few weeks from release. These types of problems are hard to diagnose ahead of time without any sort of testing regime.

One of the goals of testing is to uncover defects that will prevent a release. If these can be found early in the development cycle, the schedule can stay on track or be updated soon enough to minimize any impact.

3.2.2 Allocation of Labour

All managers need to decide how to allocate work to their employees. When the current understanding of the amount of work is wrong, this task becomes very difficult.

Testing of a program under development provides a clear view of how well it fulfills the requirements of the functional specification. It is almost like a *convolution* of the current program and the functional spec. This provides a good deal of information as to what currently works and what does not yet work. This information can be used to allow work to be allocated to actual problems.

For example, one employee may have provided all the functionality required of their section of the project, but there are still two days before the next meeting, so they add a few more "features" to their section. If a testing system was in place, the manager (or the employee himself) could see that the requirements have been met and could then move on to a section of the project that still requires work. To make an analogy to farming, it's as though you can look back at your field an see you've plowed the whole thing, but can

look over to your partner's field and see that he could use a bit of help. Without a testing system in place though, it's just guesswork about whether your code does *everything* it is supposed to.

If a manager is able to look at the test results far enough ahead of time, and see that the company does not have enough manpower to get a job done, she can use that information and outsource that work to another firm or a contractor if the work is essential. This flexibility only comes with good information provided by testing of the *current state* of a software project.

3.3 Economic

> "Because the main language of [corporate management] was money, there emerged the concept of studying quality-related costs as a means of communication between the quality staff departments and the company managers." F.M. Gryna [7]

This section will describe the costs associated with testing. These costs can be either *required* for testing to take place, or costs that are *obviated* by testing. Each section will cover the costs associated that stage of development.

3.3.1 Investment in Prevention

Prevention costs are the costs of those activities specifically suited to prevent poor quality. This covers such examples as "coding errors, design errors, mistakes in the user manuals, as well as badly documented or unmaintainable complex code."[8] Nearly all of these costs occur right in the development department, since they focus on the actual design and programming of the product.

Employee training is a very cost-effective way to reduce prevention costs. Having employees understand and practice good coding procedures such as *early prototyping* will making later testing much more useful and relevant.

The other side of reducing prevention costs is that of good design. Both experience and training can help the design team go through a requirements analysis, develop a clear specification, and then perform usability analysis. These practices are a base on which the rest of the development process follows.

3.3.2 Investment in Status Appraisal

Investment in status appraisal is spent predominantly on activities designed to find quality problems. Examples of such activities are code inspections, suites of software tests, and nightly builds. The details of these types of testing will all be covered later in Section 4.

The costs of appraising software covers employee salaries and training. When, hardware dependent software is being written, this cost also includes and hardware test tools to ensure proper operation of the device, or for troubleshooting problems. Many companies devote entire departments to testing and *quality assurance*. Smaller companies may just have a few people doing the job, as part of their normal duties.

At the Saskatoon site of PMC the investment in *status appraisal* was just a portion of time for each of the developers, and the cost of maintaining a hardware lab, with testing tools. There were regular nightly builds which took several days to set up, but completed automatically without intervention from then on.

3.3.3 Internal Costs of Failure

These are costs that occur before the product is released to customers. Many of these costs are not borne by the programming department alone, but by many other departments in the company. When a bug blocks someone else in the company from doing their job, the cost of wasted time, and overtime to get back on schedule becomes an *internal cost of failure*.

An example would be a company that needs screenshots of a program for the box that contains the program. If the user interface isn't completed on time, additional printing fees and rush charges will be added to get the job finished on time.

3.3.4 External Costs of Failure

External costs of failure are often very large. They arise after the product has been delivered to the customer and are therefore unexpected. If these costs are large enough, they can cause a project to fail or even bring down a whole company, if the one product was all they had (such as a startup). As such, they are worth considering in detail to understand what pitfalls may present themselves.

If a bug makes it into a software release, a great deal of money must be spent to find the problem and then change all aspects of the product that is affected. First, time must be taken to investigate customer complaints. This is often the way a bug is first found, and can vary in the time it takes for the problem to be found.

For example, at PMC-Sierra a large customer expressed concerns regarding sub-optimal performance in one section of a piece of software. This led to several weeks of looking through the code itself, and performing tests (to be discussed later) to narrow down the problem. In this situation there are typically only four or five customers for each product, but they each purchase millions of dollars worth of that product.

The second factor in dealing with a bug is handling support calls. In many industries the call-centre can handle these calls. However, when the customer really needs a solution, the developer (programmer) working on the actual code has to handle the call. This is especially true when both sides are trying to fix the problem, and the test results must be discussed. This type of cost is the *opportunity cost* of handling the call.

The last factor in terms of solving a software bug is reprinting manuals. This is only necessary when the bug changes the way the software works, thus changing the operating instructions. This is usually rare, but it makes it difficult to deal with customer calls when there are two versions of a manual.

Errors that occur in a pure software project (no hardware) have become more manageable to fix in recent years due to the pervasive presence of the internet. It allows updates to be downloaded both to fix errors and to enhance functionality by adding more features. However, hardware projects that are not connected to the internet do not have this advantage.

Sometimes consumer electronics ship with errors in their software. In this case, the company will have to spend a great deal of money to recall the product and fix the problem. Often the company will also have to provide refunds or discounts to keep customers from choosing a different vendor next time. The recall costs include notifying customers, shipping the replacements, and issuing refunds.

In some cases, the product will not have to be replaced entirely, but the error may just cause a decreased life-span or regular breakages. In this type of situation the company's costs will arise from increased warranty costs.

The last cost due to software errors in electronics comes from having to support *two* versions of a product in the field. Technicians will have to be trained how each works, and what is different between the two. Documentation will have to take into account the differences in each version and will have to keep them separate for the understanding of the customer.

The final type of external cost is that of legal support. When a faulty product (software or hardware) is released there is always the chance of an individual or company taking legal action to seek damages. This can be anything from lost revenue, to an unfulfilled contract, to personal injury or harm. These types of costs are potentially very large, and so every effort should be made to mitigate them.

If the bug causes a product to operate outside of legal limits, additional costs of government investigations may occur. If the problem is found, penalties and fines would also follow.

4 What can I do about it?

A *test case* is define by the IEEE Standard 610 as "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement." Essentially, a test case is meant to gather *information*. Many times this includes exposing defects (but not necessarily).

The idea that tests, once written will always retain their usefulness is specious, no matter how good it may sound. After the program has consistently passed the test many times, the information they convey drops markedly. As such, large tests can be reduced to a smaller subset of tests to allow for faster testing, while still covering the most important aspects of the program.

Tests should be appropriately complex. In the early stages of testing, simple tests are sufficient. Not only would complex tests be overkill, but in many cases they choke on *blocking bugs*. Kaner explains that "a blocking bug causes many tests to fail, preventing the test group from learning the other things about the program that these tests are supposed to expose."[2] As such, test complexity is undesirable when the program is changing in many ways, and should be saved until the program becomes more stable.

There are many types of testing that are useful in different situations. A test provides value to the extent we learn from it. A well-written test will be informative and "will teach you something (reduce your uncertainty) whether the program passes it or fails."[2] The rest of this section will describe different testing strategies that will be useful for widely separate testing needs.

4.1 Function Testing

Function testing is goes over each function/feature of a program and tests each individually. It is likely the first type of test to be written, since it covers the most basic elements of the program under test.

This type of testing is also the basis of *Test Driven Development*. The idea is that you write tests for every piece of the program before the piece is written. The process is as follows:

1. Write a test that specifies a tiny bit of functionality
2. Ensure that the test fails (you haven't built the functionality yet)
3. Write only the code necessary to make the test pass
4. Refactor the code, ensuring that it has the simplest design possible for the functionality built to date

This is a good example of a type of test that *clarifies functionality* since the test must be written before any code is developed.

These tests aren't expected to fail, unless the algorithm is fundamentally flawed. As such, they can catch problems right away when changes are being made to the structure of the code. They are also easy to evaluate, since each test covers only one aspect of a function.

Most functional tests are written as "black-box tests working off a functional specification."

4.2 Domain Testing

Domain testing is similar to *function testing* but uses a range of values for each variable under test rather than just ensuring each function works.

When the range is chosen, it should include value on both sides of the acceptable limits (boundaries). Therefore, many of these tests are run to ensure that the program fails where it should (for values beyond the limits). Kaner explains that "A good set of domain tests for a numberic variable hits every boundary value, including the minimum, the maximum, a value barely below the minimum, and a value barely above the maximum."[2]

Test frameworks such as the open-source DejaGnu allow for tests to have *expected fail* as the presupposed result of tests. The same result can be achieved with other testing frameworks by *assert*ing that a given function will raise a certain exception. Both of these procedures essentially test to make sure that a given test fails.

Domain tests carry significant *information value* when they are first run. This is because boundary and extreme-value errors are very common. They often reveal flaws in the design of the algorithm under test.

4.3 Specification-based Testing

Specification-based testing uses tests to check for specific claims made for a reference document.

This type of testing is very important when outsourcing or contracting work to a third-party. There will be a design specification referenced in the contract signed by both parties and every feature specified must be tested. One party (usually the contractor) will have to create a set of tests that shows that each feature works. We read that "a good set of tests includes an unambiguous and relevant test for each claim made in the spec." This shows that they have kept their side of the contract and is often a prerequisite for payment.

A good deal of the software and firmware development at PMC was contracted out to third parties. Before the requirements of the contract could be fulfilled, a PMC employee would go to the contractor and perform *Field Acceptance Testing*. Each requirement in the design specification would be tested and the result recorded. Although the tests were not always automated, the same process was followed as if they were.

4.4 Regression Testing

A *regression* is a "relapse to a less perfect or developed state."[4]

4.4.1 Nightly Builds

Nightly builds are a very common type of regression test that run a series of tests on the latest version of a program.

The tests are often run at night for two reasons. First of all, most employees have gone home to sleep leaving the processing power of any mainframes available. On one hand, this benefit has been increasing lately because of the grid computing initiatives many IT departments have been implementing. This means that even regular workstations can be used for computation while they are idle. On the other hand, increasing internationalization of companies has meant that night-time at one location is the middle of the day in another. As such, any downtime is just used up by other sites.

The tests are run at night, secondly, for the reason that it allows developers to check out working copies at the beginning of the day, add their changes, and then make sure everything works by the time they leave at the end of the day. This factor has lost much of its pull with the use of *revision control systems* which allow developers to *check out* a working copy of the code, then work on it all they want, then check their changes back in once all their changes work again. In essence, it gives the programmers as long as they want to work on their features without breaking the build for anyone else.

PMC-Sierra has regular nightly builds which compile each program under development, and email the results out. This system allows developers to know by the next if something caused to program to stop compiling. In that regard it's very good. On the other hand, the tests stop transmitting much information once the tests have all passed for three weeks in a row. Even if one of the tests fail, the email does not give any indication as to where the problem was.

One of my projects was to take the log-files created by these tests and make them more usable. I started by organizing the log-file data into a special file (XML in this case). I then built a website which showed all the tests which had been run in the last few months (using data from the XML file).

4.4.2 Check-In Testing

Many revision control systems **??** allow a script or program to be run when new changes have been made. This is similar to nightly builds in that regular tests are being run, but it has an advantage that the developer who made the changes sees the test results right away, rather than the next morning.

Check-in tests are advantageous for international software companies where employee workdays at different locations overlap due to the different time-zones. In such a case, the invocation of tests immediately upon the submission of changes is ideal.

4.4.3 Hardware

For tests requiring hardware, special modifications must be made to allow it to be automated. A software access layer will need to be used. This could be an API for a specific programming language or a standalone program to control hardware switches.

Serial relay boards are a very popular option. They use a very common connector and often ship with a set of drivers to open and close the relay contacts.

The regression test system at PMC has a host computer connected to a rack filled with evaluation boards with the new microchips on them. The power is run through a relay board which is, in turn, controlled by the host computer. Each board can be turned on individually with the software tool through a hardware connection to the power-switch circuitry.

4.5 High-volume Automated Testing & Fuzz Testing

Many software errors can be found with mere functional and domain testing. However, some errors are more insidious and do not show up unless the program or network is under high load or receives spurious inputs.

High-volume testing is similar to domain testing in that the range of values changes to test different cases of a program. However, in this case the range is of an orthogonal set of information. Instead of change the input to be a range of values, the input becomes a range of input *volumes*. In other words, that rate at which the program handles input is changed, to ensure that is can handle ranges up to what was specified in the functional specification.

This type of testing can also be used with a set volume of data being left on for a stated length of time. This will help to check for memory leaks. Since they are so small, having the test run for an extended length of time can uncover errors that were previously not observed.

High-volume testing is commonly used to test network applications (both high and low-level) because their input often varies widely. For example, at PMC-Sierra we had an ethernet driver that needed to be tested for errors. A test was set up to flood the router with data constantly all night (or all weekend in some cases). This allowed errors that occurred rarely to still be observable. In one case, the error took several days of running before it would show up. In this case, there would have been no other way to discover this error.

A related type of testing is called *Fuzz Testing*. It sends random data to the inputs of a program to try to expose defects. This type of testing often reveals unwanted results due to input that the programmer would have never anticipated. Since the testing is so simple (nothing needs to be known about the internals of the program), it has a high benefit to cost ratio. As such, it is a very common type of test for finding defects.

5 Conclusion & Recommendations

In many cases, software testing can (and should) be automated. The problem is manual testing is that it takes a long time, and therefore doesn't get done. If an organization only takes one step forward and just writes tests, the same problem remains, it takes a long time. When tests are automated, they *will* get run, and developers can use the extra time to fix errors, and write more detailed test cases.

The goal of this paper was to provide a cogent analysis of software testing. However, no discussion spent on either the creation of Functional Specifications nor on writing good Test Cases. These are each very large topics, and are very valuable for implementing a testing framework.

This discussion may have sounded like it was merely pontificating. However, when these ideas are put into practice it can be seen that they have a very pragmatic appeal.

References

[1] Ram Chillerege, "Software Testing Best Practices," Center for Software Engineering, IBM Research, April 1999

[2] Cem Kaner, "What Is A Good Test Case?" Department of Computer Science, Florida Institute of Technology, May 2003

[3] Rob Savoye, "DejaGnu - The GNU Testing Framework" Free Software Foundation, 2004 *http://www.gnu.org/software/dejagnu/manual/*

[4] "Canadian Dictionary of the English language" International Thompson Publishing, 1997

[5] J.-C. Laprie at al., "Dependability: Basic Concepts and Terminology," *Dependable Computing and Fault-Tolerant Systems, 1992*

[6] Daniel Siewiorek, Rame Chillarege, Zbigniew Kalbarczyk, "Reflections on Industry Trends and Experimental Research in Dependability" IEEE Transactions on Dependable and Secure Computing, Vol 1, No 2, June 2004

[7] J.M., F. M. Gryna, "Quality Costs in Juran" Juran's Quality Control Handbook, McGraw-Hill

[8] Cem Kaner, "Quality Cost Analysis: Benefits and Risks" Software QA, Volume 3, #1, 1996